Finite domain constraint solving

Broek, J.M.

Publication date:
1990

Document Version
Publisher's PDF, also known as Version of record

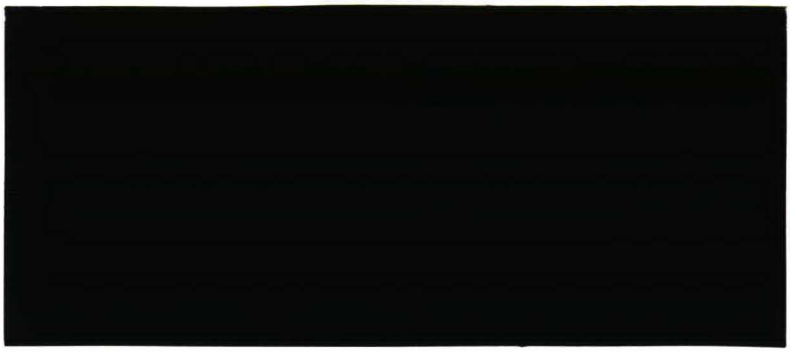
E.T

ITK

MEMO

INSTITUTE FOR LANGUAGE TECHNOLOGY AND ARTIFICIAL INTELLIGENCE

INSTITUUT VOOR TAAL- EN KENNISTECHNOLOGIE



I.T.K. Research Memo no. 5
August 1990

Finite domain constraint solving

Johan M. Broek

Abstract

In this document an overview of constraint solving techniques for constraint satisfaction problems is given. It is shown that many of these techniques have a common basis. Starting from simple methods we show how several techniques can be combined to yield powerful constraint solvers. Some of this is illustrated in a simple constraint solver written in Scheme.

Contents

Chapter 1

Introduction

Constraint is an essential notion in AI. The terms constraint, constraint satisfaction, and constraint solving, however, have been used in many different contexts and meanings. In this paper we try to identify the common principles. Our aim is to demonstrate that it is feasible to build general and extensible constraint solvers for constraint satisfaction problems (CSPs) [1] that maintain a high level of efficiency. Starting from simple methods we show how several techniques can be combined to yield powerful constraints solvers. Here we pay special attention to design decisions. To keep the discussion rather simple we will illustrate these ideas by means of two simple and well known AI puzzles, the n-queens problem and the Send + More = Money problem, and some programs written in Scheme.

1.1 Constraints

A constraint describes a relation that is either true or false, that is, satisfiable or unsatisfiable. A few examples of constraints:

- The age of the candidate should not exceed 21.
- $v_2 \in D_4$.
- The word starts with an 'f'.
- 'Everybody loves my baby'
- 'My baby loves nobody but me'

[1] That is, *finite domain* constraint satisfaction problems.

By combining constraints, using logical operators, complex constraint problems can defined. A computer program that determines whether or not a constraint problem has a solution, is called a *constraint solver*. Examples of such solvers are Simplex or unification algorithms.

An important observation is that problem formulation and problem solving are strictly separated. Indeed, the constraints (usually) only represent declarative knowledge. The constraints are used to formulate the problem, and the constraints solver computes one or more solutions of the problem, if any exist. Stating a problem as a set of constraints yields several advantages:

- the constraints state properties directly in the intended domain of discourse [23];
- the constraints have the ability of representing properties implicitly as opposed to having bindings to variables;
- the constraint solving paradigm enables the use of interesting problem solving techniques like local value propagation, data-driven computation and consistency checking [39].

1.2 Applications & Programs

The last two decades much has been written about constraint solving and constraint satisfaction. Some of the early papers are [40], [34] and [38]. We will confine attention to a special class of constraint problems: finite domain constraint satisfaction problems, although a short overview of other approaches is given. Instead of jumping into technical details, we would like to illustrate [2] the versatility of the constraint approach by describing several areas of application.

1.2.1 Design

Sketchpad [38], is a constraint system created by Sutherland at MIT in the early 1960s. It allows the user to build geometric figures using primitive graphical entities and constraints. Using Sketchpad a user draws a complex object by sketching a simple figure and then adding constraints to it. Examples of such constraints are, making two lines parallel, perpendicular, or of equal length. Sketchpad satisfies constraints by using a one-pass method

[2] We do not give an exhaustive overview.

(propagation of degrees of freedom), or if the one-pass method fails, relaxation is used. It was one of the first systems based on constraints and was in many ways revolutionary. Unfortunately, it needed quite some CPU time, and special display hardware, which were far too expensive in those days.

Many years later Borning begat ThingLab [4]. It is based on ideas in Sketchpad combined with the extensibility of the object-oriented techniques of the Smalltalk programming language. In addition to the constraint satisfaction methods used by Sketchpad, ThingLab also uses "propagation of known states". Apart from geometric layout, it has been used to simulate the behavior of physical systems.

Other design systems based on constraints include Margritte by Gosling [19], ThingLab II [17], and Ideal by Van Wyk [41]. Somewhat related is Knuth's Metafont program [25].

1.2.2 Simulation

A natural application area is the simulation of physical and economical systems. Constraints can be used to represent physical or economical laws, causal relations, or definitions.

Stallman and Sussman used constraints to simulate electrical circuits [35]. Their system (EL and ARS) is based on ideas in Sketchpad. Values of known variables are 'propagated trough the network of constraints' using forward reasoning rules to compute the values of unknown variables. If an inconsistency is encountered the program backtracks using a technique known as dependency directed backtracking.

Qualitative physics, an active area of research in AI, is concerned with qualitative rather than quantitative analysis and simulation of physical systems [3]. Most systems, in the area of qualitative physics, employ some sort of constraint mechanism [14, 26]

In qualitative economics, one tries to model and explain economical systems on the basis of qualitative data. Qualitative constraints are used to state book keeping relations or economic laws [1]. The constraint solvers are usually based on CSP techniques, as will be described in chapter 2.

In [5], the authors describe an asset & liability management (ALM) simulation model implemented in the constraint logic programming language Chip. Constraints are used to model the behavior of several bank control accounts subject to policy rules, changes in interest rates, and optimization objectives. The program uses a simplex-like algorithm in combination with tree search techniques and heuristics to find interesting ALM strategies.

1.2.3 Diagnosis

Like simulation, diagnosis is a natural application domain of constraint based systems. Indeed, often simulation models are used within diagnostic systems. Examples of constraint-based diagnostic systems can be found in [13] and [9].

Logisim [20], is a program for simulating and diagnosing hybrid circuits. These hybrid circuits, which are widely used in the aircraft industry, consist of electromechanical, electrohydraulical, and hydromechanical components. Logisim employs several constraint solvers to validate these hybrid circuits.

Feelders [16], reports the use of constraint solving techniques for financial diagnosis. The system uses a combination of quantitative and qualitative information, to identify and explain significant changes in the financial structure of a firm.

1.3 Constraint languages

Recently, people have started to investigate general-purpose languages based on constraints. Constraints, is such a language, and was designed by Steele as part of his PhD thesis [37, 36]. It can be seen as an extension and generalization of the work of Stallman and Sussman on El and ARS. The system maintains dependency information to support dependency directed backtracking and the generation of explanations. Like El, Constraints has been mainly used in the simulation of electrical circuits.

Bertrand, is a general purpose programming language based on augmented term rewriting [28]. Bertrand is not a constraint language, but rather a tool for implementing new constraint-based systems. It has been applied to small examples in graphics and electrical circuits.

Of growing importance is the integration of constraints solving techniques and logic programming to yield constraints logic programming systems. These general purpose languages combine the declarative aspects of logic programming with the efficiency of constraints solving techniques. Examples of constraints programming languages are CLP($\mathcal{R}$) [24], Chip [39], and Prolog III [7]. The constraint logic programming scheme (CLP), defines a class of languages based upon the paradigm rule-based constraint programming [23]. Each instance of the scheme is a programming language and is obtained by the specification of the domain of computation. A domain of computation is defined by:

- the basic elements in the domain, e.g., all integers;
- the allowed operators on the basic elements, e.g., $+, *$;
- the predicates on terms constructed of basic elements and operators, e.g., $=, \neq, \geq$.

A particular instance of the CLP scheme is CLP($\mathcal{R}$). The domain of computation is the set of real numbers and the constraints solved by the constraint solvers are linear equations and inequations.

Prolog III, uses a simplex like algorithm to solve linear equations and inequations of rational terms, and provides a saturation method to deal with Boolean terms.

Chip provides three computation domains: finite domain restricted terms, Boolean terms and rational terms. For each of them Chip uses specialized constraint solving techniques: consistency techniques [3] for finite domains, equation solving in Boolean algebra for Booleans and a symbolic simplex-like algorithm for the rationals.

Summary and notes

Constraints have been studied for three reasons. Firstly, constraints can be used to represent knowledge. Secondly constraints form a computational paradigm. Finally, the constraint approach enables efficient control strategies. Constraints can be used in a variety of problem domains. Although we did name some constraint solving techniques that are used, we did not define them. Some of these techniques will be described in the next chapter. The interested reader can trace the references for more information on the other techniques.

[3] essentially CSP methods, see chapter 2.

Chapter 2

Constraint solving

2.1 CSPs

A constraint satisfaction problem (CSP) is characterized by a set of variables. Each variable has a domain of values and a set of domain constraints. A solution to a CSP assigns a value consistent with the domain constraints to each variable.

In the following we assume that all domains are closed and contain only a finite number of elements. In other words, every domain has a cardinality. For this reason we will also refer to CSP solvers as *finite domain constraint solvers*. Definition (Mackworth [31]):

> A Boolean Constraint Satisfaction Problem (CSP) is specified if we have a set of variables
>
> $$V = \{v_1, \ldots, v_n\}$$
>
> and a set of constraints limiting the set of allowed values for specified subsets of the variables. Each variable takes on values in some domain. The set of solutions to the CSP is the largest subset of the Cartesian product of the domains of the n variables such that each n-tuple in the set satisfies all the given constraint relations.

In the previous chapter it has been illustrated that the CSP paradigm allows modeling of a wide range of 'real life' problems. In this chapter, we will apply the CSP paradigm to AI puzzles. These puzzles are rather simple and therefore allow us to solely concentrate on constraint solving techniques.

One of the drawbacks of solving puzzles, instead of 'real life' problems, is that solving puzzles is not enough to convince people that you can solve their real problems. Indeed, an important conclusion of many years of AI research is that solving small puzzles is not enough. Firstly, many problems are usually encountered when trying to extend the solution of the puzzle to 'real size' and 'real life' problems. On the other hand, if our system cannot solve these two AI puzzles then surely it will not solve the real problems.

Example: the n-queens problem

Given sets $C = \{1,\ldots,n\}$, $R = \{1,\ldots,n\}$, and $S = \{-1,0,1\}$, find a function $f : C \rightarrow R$ such that,

$$\forall x \in C > y \in C, \forall a \in S : f(x) \neq f(y) + ax - ay.$$

In English: Place n queens on an $n \times n$ chess board in such a way that they do not attack.

Example: send + more = money

Given two sets $V = \{s,e,n,d,m,o,r,y\}$ and $L = \{0,\ldots,9\}$, find a function $f : V \rightarrow L$ with the following properties, $f(s) \neq 0$, $f(m) \neq 0$,

$$f(s) \neq f(e) \neq f(n) \neq f(d) \neq f(m) \neq f(o) \neq f(r) \neq f(y),$$

and

$$\begin{array}{rcrcrcrcl} & & 1000f(s) & + & 100f(e) & + & 10f(n) & + & f(d) & + \\ & & 1000f(m) & + & 100f(o) & + & 10f(r) & + & f(e) & = \\ 10000f(m) & + & 1000f(o) & + & 100f(n) & + & 10f(e) & + & f(y). & \end{array}$$

In other words, instantiate the variables in V using values in L. This instantiation should satisfy the equation Send + More = Money, and all variables should have a different value.

2.2 Free/ground paradigm

In a CSP the aim is to assign values, from the domains, to the variables, consistent with all the domain constraints.

A (domain) variable consists of two parts: a domain of values and a value. The value of a variable is either known or unknown and should be in its domain.

Let **free** be a set that contains all the variables with unknown values, and **ground** a set of variables with known values. Clearly, the intersection of **free** and **ground** is empty. All the variables in **free** are called *free* or ***unknown*** and all the variables in ground are called ***ground*** or ***known***. If all the variables are in ground then we call ground a ***full instantiation***. A full instantiation that satisfies all the domain constraints is a ***full solution***. Stated differently, the goal in constraint solving is to move all variables from **free** to **ground**, in such a way that **ground** is a full solution.

2.3 Generate & Test

A simple method to solve the two examples is based on explicit enumeration of full instantiations. To compute solutions of the 4-queens problem, for instance, one could generate all full instantiations, and test which are solutions. This method is known as *generate & test*.

Example: n-queens first try

The test for the n-queens problem is a predicate which is defined by the following function.

```
(define no-attack?
  (lambda queens
    (define safe?
      (lambda (one others up)
        (or (null? others)
            (and (not (= one     (first others)   ))
                 (not (= one (+ (first others) up)))
                 (not (= one (- (first others) up)))
                 (safe? one (rest others) (1+ up))))))
            (or (null? queens)
                (and (safe? (first queens) (rest queens) 1)
                     (apply no-attack? (rest queens)))))))
```

The predicate **no-attack?** is *true* if no queen attacks other queens, and *false* otherwise. Furthermore, the function

```
(join domain*)    ;;; domain* denotes "zero or more domains"
```

computes the Cartesian product of a number of domains. We refer to an element of the Cartesian product as *tuple* [1]. So, `join` returns a set of tuples. Finally the function,

```
(define select
  (lambda (predicate set)
    (cond ((null? set)
           '())
          ((apply predicate (first set))
           (cons (first set)
                 (select predicate (rest set))))
          (else (select predicate (rest set))))))
```

selects all tuples from `set`, for which the `predicate` is true. The `queens` function computes all solutions of the n-queens problem.

```
(define queens
  (lambda domains
    (select no-attack?
            (apply join domains)))).
```

If this function is called with the following input,

```
(queens '(1 2 3 4) '(1 2 3 4) '(1 2 3 4) '(1 2 3 4)),
```

the result of applying `join` will be,

```
((1 1 1 1) (1 1 1 2) ... (4 4 4 3) (4 4 4 4)),
```

and `select` returns,

```
((2 4 1 3) (3 1 4 2)).
```

Generate & test can be generalized easily. Let `csp` be a constraint satisfaction problem, and `test` a predicate that returns true if its input is a full solution of `csp`, and false otherwise. The following function computes all the solutions of `csp`.

[1] A tuple in the Cartesian product of all the domains in a CSP, is, of course, a full instantiation

```
(define csp
  (lambda domains*                    ;;; Zero or more domains
    (select test
            (apply join domains*)))
```

Generate & test is very simple, but also very inefficient. To solve the n-queens problem, for example, n^n tuples will be generated and tested. In general, if the CSP has n variables and every variable has a domain of cardinality c then c^n tuples will be generated and tested. Clearly better methods can be found.

2.4 Choice & Backtrack

2.4.1 Partial solutions

Consider the tuple `(? ? 1 2)`, where the variables in position 1 and 2 are still unknown. No tuple with this pattern in the last two positions can be a full solution of the 4-queens problem, because the queens in position 3 and 4 attack each other. On the other hand, we cannot say, a priori, whether or not the *partial instantiation*, `(1 3 ? ?)`, can be extended to a full solution. In other words, we are not able to check sufficient conditions. However, we can enforce necessary conditions on partial instantiations, and thus avoid generation of some useless full instantiations. We refer to these necessary conditions as *partial test*. A *partial solution* is a partial instantiation that satisfies a partial test. A partial solution may perhaps be extended to a full solution. However, partial instantiations that do not pass a partial test can never be extended to a full solution.

2.4.2 Standard backtracking

Recall the free/ground paradigm. The goal was to move all variables from `free` to ground in such a way that ground defined a full solution. Using the generate & test method we first moved all variables from `free` to ground at once, and then applied a full test. Now we define a procedure `choice`, that chooses a variable from `free`, assigns a value from the domain of the variable and adds it to ground. We also define a second procedure `backtrack`, that chooses a variable from ground and moves it to free. Roughly, `backtrack` does the opposite of `choice`. Finally we define a procedure `partial-test` to test the partial instantiations.

So far we did not define a choice or backtrack order, and in fact we don't have to. In the following, however, we assume that the variable selection order of `backtrack` and `choice` are fixed. Moreover, we assume that the domains are ordered and `choice` always assigns values in order.

Assume all the variables are stored in a list

$$(v_1, \ldots, v_i, \ldots, v_n),$$

where variables $v_1, \ldots, v_{i-1}$ are in ground and $v_i, \ldots, v_n$ are free. We assume that the assignment of values is a partial solution. Let v_i be the variable in `free` with the highest index. We call v_i the *current variable*, or *choice point*.

We have defined a lot of things, and now we put some of it together. The following algorithm tries to find and extend partial solutions in order to find a full solution.

1. `first choice`. If `free` is empty, then stop and present the solution, else let v_i be the current variable. Remove v_i from `free`, assign the first value in the domain of v_i, to it, and move v_i to `ground`.

2. `test`. Apply `partial-test` to `ground`. If `partial-test` returns true then go to `first choice`, else

3. `retry choice`. If `ground` is empty, then stop and report a failure, else let v_i be the variable in `ground` with the lowest index. Assign the next value in the domain of v_i, and `test`. If v_i has no more untried values left in its domain, then `backtrack`.

4. `backtrack` Let v_i be the variable in `ground` with the lowest index. Move v_i from `ground` to `free`. Go to `retry choice`.

The search method sketched above is called depth first search with chronological backtracking, or *standard backtracking* for short.

Example: n-queens revised

The queens* program employs a standard backtracking algorithm, in order to find a stream of solutions of the n-queens problem. Because we assume that both the choice and backtrack order are fixed we don't have to represent the variables explicitly.

```
(define value car)
```

```
(define untried cdr)

(define queens*
  (lambda (n)
    (let ((domain (enumerate-interval 1 n)))
      (define build-solution
        (lambda (ground)
          (if (= n (length ground))
              (cons-stream (map value ground)
                           (backtrack ground))
              (choice domain ground))))
      (define choice
        (lambda (current-choice ground)
          (cond ((null? current-choice)
                 (backtrack ground))
                ((partial-solution? (value current-choice)
                                    ground
                                    1)
                 (build-solution (cons current-choice
                                       ground)))
                (else (choice (untried current-choice)
                              ground)))))
      (define backtrack
        (lambda (ground)
          (if (null? ground)
              the-empty-stream    ;;; no more solutions
              (choice (untried (first ground))
                      (rest ground)))))
      (build-solution '()))))
```

The partial test is almost the same as the `safe?` predicate of the generate & test version of n-queens.

```
(define partial-solution?
  (lambda (one others up)
    (or (null? others)
        (and (not
              (= one
                 (value (first others))))
             (not
```

```
        (= one
           (+ (value (first others))
              up)))
        (not
         (= one
           (- (value (first others))
              up)))
        (partial-solution? one
                           (rest others)
                           (1+ up))))))
```

This function generates a list all integers from start to end.

```
(define enumerate-interval
  (lambda (start end)
    (if (> start end)
        '()
        (cons start
              (enumerate-interval (1+ start)
                                  end)))))
```

The behavior of the standard backtracking version, queens*, is a little harder to describe than the behavior of queens, (the generate & test version). Let us trace a few steps of the program, when solving the 4-queens problem.

First, build-solution is called with input (). As the length of ground is equal to zero, we haven't found a solution yet. So, it calls choice:

```
(choice '(1 2 3 4) '()),
```

and choice calls the partial test.

```
(partial-solution? 1 '())
```

Yes, a chess board with only one queen is a partial solution. So, choice can continue.

```
(choice '(1 2 3 4) '((1 2 3 4)))
```

Again we apply the partial test.

```
(partial-solution? 1 '((1 2 3 4)))
```

No, bad luck, try the next value.

```
(choice '(2 3 4) '((1 2 3 4)))
```

Okay, let's try the next one.

```
(partial-solution? 2 '((1 2 3 4)))
```

Sorry, try the next one (we will skip the choice part from now on).

```
(partial-solution? 3 '((1 2 3 4)))
```

Hurray, we have found the next partial solution. Continue.

```
(partial-solution? 1 '((3 4) (1 2 3 4)))
```

Don't worry, try the next one.

```
(partial-solution? 2 '((3 4) (1 2 3 4)))
```

Well okay, the next one perhaps?

```
(partial-solution? 3 '((3 4) (1 2 3 4)))
```

Ah well, one more try left.

```
(partial-solution? 4 '((3 4) (1 2 3 4)))
```

Hmmmm. No more tries left.....Backtrack!

```
(backtrack '((3 4) (1 2 3 4)))
```

Backtrack calls choice.

```
(choice '(4) '((1 2 3 4))
```

The program proceeds until the first full solution has been found, and returns a pair. The car of this pair is the first solution, and the cdr of this pair is a delayed function that can generate the next solution.

2.4.3 The pros and cons of standard backtracking

Although not a simple as generate & test, standard backtracking is quite a simple algorithm to implement and optimize. It is for this reason that standard backtracking has been used in many programs. Prolog, for instance, uses a standard backtracking algorithm. For the same reason we will use the standard backtracking as a basis for all constraint solvers described in the following sections.

However, standard backtracking is still far from the desired result. It suffers from behavior known as *thrashing* [30, 39]. Thrashing is manifested by the following symptoms.

- Bad backtracking point; the procedure backtracks to the most recent choice which has probably nothing to do with the current failure.
- Late detection of failures and useless generation; the failures are detected late in the search and after some useless generation, thus increasing the amount of backtracking.
- Continual rediscovery of the same facts; the fact that some values satisfy or do not satisfy a particular constraint is rediscovered many times during the search.

Better results can be obtained by using heuristics to determine which variable to choose, which value to assign, and to which choice point to backtrack.

2.4.4 Dependency directed backtracking

Dependency directed backtracking [35], backjumping [10], or intelligent backtracking [6] are different names that all stand for the same technique. Both in the logic programming community and constraint programming community quite some effort has been devoted to improving backtracking. Standard backtracking always backtracks to the most recent choice point regardless of the cause of the current failure. Dependency directed backtracking is a method that tries to avoid retrying unproductive choice points by analyzing the source of the current failure. This source can be found by analyzing the dependencies between the variables and the constraints [11, 10, 35]. Dependency directed backtracking has been applied successfully in many systems [35, 36, 15]. Its counterpart in logic programming, however, seems to be less successful. Reasons for this limited success are that you need special

heuristics to find the best backtracking point [2] and that dependency directed backtracking induces overhead for programs not using the technique. As has been said before in the following we will mainly consider standard backtracking.

2.4.5 Choice heuristics

Another method to improve the efficiency of the standard backtracking algorithm is to use heuristics to instantiate variables in the right order with the right value. Examples of such heuristics are:

1. choose the variable with the smallest domain,
2. instantiate the most constrained variable,
3. assign the most constrained value, or
4. cutset decomposition [10].

The first three heuristics are based on empirical observations. The basic idea behind these heuristics is: "To succeed try first where you are most likely to fail." The last heuristic is a bit more tricky. As stated before many CSPs are NP-complete. This means that a polynomial algorithm to solve them is unlikely to be found. However, some CSPs can be solved using a polynomial algorithm. Consider a (hyper) graph where the domain variables are the edges and the nodes are the constraints of the CSP (or the other way around). If this graph is a tree then the CSP can be solved in polynomial time. The cutset decomposition method tries to transform the constraint graph of the CSP to a tree by cutting all the cycles in the graph. The cutset is the set of all variables that if instantiated cut the cycles in the constraint graph. The remaining CSP, can solved in polynomial time.

[2] This comparison is not very fair. Backtracking algorithms in logic programming, as exemplified by Prolog, need to be much more sophisticated then the ones described above. We have used the term 'choice point' because the `choice` procedure can assign different values from the domain of a variable, which is a rather simple matter. In logic programming, however, it refers to the situation where the resolution algorithm, say SLD resolution, can use several clauses to perform a goal substitution, which is a far more complicated matter.

2.5 Local propagation

2.5.1 Value propagation

One of the thrashing symptoms of standard backtracking is the late detection of failures and useless generation. *Local value propagation* algorithms try to compute the values of free variables using the values of the known variables [4, 36], in order to detect failures as soon as possible. Consider the following example:

$$\begin{aligned} a + b = c \quad & \wedge \\ c + a = d \quad & \wedge \\ a \in \{1,2,3\} \quad & \wedge \\ b \in \{1,2,3\} \quad & \wedge \\ c \in \{1,2,3\} \quad & \wedge \\ d \in \{1,2,3\} & \end{aligned}$$

If a and b are ground and c is free, then the value of c can be inferred from the semantics of the constraint. Suppose $a = 1$ and $b = 1$, then we can safely conclude that $c = 2$. Now c can be moved to ground. Furthermore, we can compute the value of d. Standard backtracking, however, first tries $c = 1$, $d = 1$, and $d = 2$ before it finds the correct solution.

We would like to extend the standard backtracking algorithm with local value propagation. For the sake of simplicity we will assume that every n-place constraint c_i can be written as n constraint rules [3]. A constraint rule r_{ij} is defined by pair (P_{ij}, f_{ij}) where antecedent P_{ij} is a predicate on *antecedent variables*, and f_{ij} is a function that computes the **value** of single variable, called the *consequent variable*, given the **values** of the antecedent variables. Every n-place constraint,

$$c_i(v_1, \ldots, v_n),$$

can be transformed to a number of constraint rules with corresponding functions such that,

$$\begin{aligned} v_1 &= f_{i1}(v_2, \ldots, v_n) \\ v_2 &= f_{i2}(v_1, v_3, \ldots, v_n) \\ &\vdots \\ v_n &= f_{in}(v_1, \ldots, v_{n-1}), \end{aligned}$$

If a constraint rule is applied and its antecedent holds, we say that it *fires*.

[3] This incorrect assumption will be revised later on.

Example: a constraint rule

Consider the following functions. The predicate

```
(ground? variable)
```

returns true, if its argument `variable` is a known variable, and false otherwise. The function

```
(val variable)
```

returns the value of `variable`. Finally the function

```
(assign variable value)
```

tries to bind `variable` to `value`. Thus, `assign` moves `variable` from `free` to `ground`. `Value` should, of course, be in the domain of `variable`. If `variable` is already ground, the `assign` checks if the value of `variable` is equal to `value`. If the values are different then `assign` reports a failure.

```
(define rule-34
  (lambda (v1 v2 v3 v4)
    (if (and (ground? v1)
             (ground? v2)
             (ground? v3))
        (assign v4 (+ (val v1) (val v2) (val v3)))
        'the-rule-did-not-fire)))
```

In the above (pseudo code) example `v1,v2,v3` are the antecedent variables and `v4` is the consequent variable.

Naive value propagation

A naive method to organize the application of the constraint rules is to try the rules sequentially. Let us call this algorithm `A-1`. `A-1` repeatedly tries all rules to see if one or more rules fire.

- `first rule`. Try the first rule (r_{11}).
- `determine`. If the rule does not fire, then go to `next rule`, else three things can happen.
  1. The constraint rule adds a new variable to `ground`. In that case, go to `first rule`.

2. The constraint rule computes a new (different) value of an already known variable. Thus an inconsistency has been detected. `A-1` halts and `backtrack` takes over control.
3. The consequent variable is already known and its value is the same as the value computed by the constraint rule.

- **`next rule`.** Try the next rule, and go to **`determine`**. If there is no next rule, then stop and let `choice` proceed.

If the local propagation algorithm stops, and no inconsistency has been detected, we call ground *locally consistent.* Moreover, if all variables are ground then we call ground globally consistent.

We need to change `backtrack` a little. If `A-1` reports a failure, then `backtrack` moves all variables from ground to `free` that have been added by `A-1` since the last choice [4] The local propagation algorithm (`A-1`) does not create choice points. `Backtrack` removes all variables from ground that have been added since the most recent choice.

Data-driven value propagation

`A-1` is an inefficient algorithm. It is not very smart to scan the whole list of constraints rules if only one variable changes at the time. Fortunately, the behavior of the local propagation algorithm can be improved without difficulty.

Let depend be a set of D_i, where D_i is a set of constraint rules, with the property that, $r_{pq} \in D_i$ if and only if v_i is an antecedent variable of constraint rule r_{pq}. We refer to D_i as the *dependency set* of v_i. Let agenda be a proper set.

The new local propagation algorithm `A-2` is different from `A-1` in the following respect:

1. **`shedule`.** If a variable v_i, is added to ground look up its dependency set D_i, and add all rules in D_i to agenda.

[4] A chronological backtracking algorithm can be implemented easily using a stack. Adding a variable to ground, then becomes the same as pushing the variable on the stack. Removing a variable is the same as popping the stack. All we have to do is to mark all the variables that have been added by choice as choice points. The backtracker pops the stack until to top of the stack is a choice point. Which, by major coincidence, also is the most recent choice point.

2. **apply**. If **agenda** is empty, then go to **choice**, else remove a constraint rule from the **agenda**. If the rule fires and succeeds, i.e., a variable is added to **ground**, go to **shedule**; else if the rule fires but fails, i.e., an inconsistency is found, then remove all the constraints rules from the **agenda** and **backtrack**; If the rule does not fire or it computes an already known and consistent value then repeat **apply**.

The efficiency of the A-2 algorithm can be improved significantly by assigning priorities to constraint rules. Remember, constraint rules can do two interesting things: compute a new value or detect an inconsistency. If **agenda** contains a constraint rule that may possibly detect an inconsistency, then we want this rule to be 'triggered' before all the other rules. Indeed, deriving new values from the known but inconsistent set of values is of no use. Also, our assumption that every n-place constraint can be transformed to n constraint rules is incorrect. As this issue is rather important we study it in some detail.

Let c_i be an n-place constraint on variables $v_1, \ldots, v_n$ with corresponding domains $d_1, \ldots, d_n$. Let d be a domain with the property that all $d_i \subset d$. The constraint c_i is a function

$$c_i : P \rightarrow \{0, 1\},$$

where P denotes the Cartesian product $d_1 \times \cdots \times d_n$. The set T is defined as follows.

$$T = \{x \in P | c_i x = 1\}$$

Thus T is the set of all consistent tuples. We call c_i a *positive constraint* if there exist n functions,

$$\begin{array}{lll} f_{i1}: & d_2 \times \cdots \times d_n & \rightarrow d \\ f_{i2}: & d_1 \times d_3 \times \cdots \times d_n & \rightarrow d \\ & \vdots & \\ f_{in}: & d_1 \times \cdots \times d_{n-1} & \rightarrow d, \end{array}$$

that satisfy the relation,

$$\forall (l_1, \ldots, l_n) \in T :$$
$$(f_{i1}(l_2, \ldots, l_n), f_{i2}(l_1, l_3, \ldots, l_n), \ldots, f_{in}(l_1, \ldots, l_{n-1})) = (l_1, \ldots, l_n).$$

In other words, if two n-ary tuples in T have $n-1$ variables in common then they are identical. Constraints that are not positive are called *negative*.

To be able to handle negative constraints we distinguish two types of constraint rules: negative and positive. In the previous section we already described the positive constraint rule. A negative constraint rule is an antecedent-consequent pair where the antecedent is a predicate that returns true if all its arguments are ground and the consequent is a constraint that returns true if its input values are consistent. Notice that negative constraint rules are almost the same as normal constraints except they have a predicate which tells when they can be applied. Furthermore, as negative constraint rules do not compute new values there is no need to make a distinction between antecedent and consequent variables. That is, all the variables in a negative constraint rule are antecedent variables.

Example: A negative constraint rule

```
(define !=          ;;; not equal
  (lambda (v1 v2)
    (if (and (ground? v1)
             (ground? v2))
        (not (=  (val v1) (val v2)))
        'the-rule-did-not-fire)))
```

We now define the `A-3` algorithm. It is almost the same as `A-2` with the following exceptions. The agenda consists of two subsets: `negative` and `positive`.

1. `shedule`. If a variable, v_i, is added to `ground`, then look up its dependency set D_i, add all negative constraints in D_i to `negative`, and all the positive constraints to `positive`.

2. `apply`. First try all the constraint rules in `negative` before it trying the constraint rules in `positive`. If both sets are empty, then go to `choice`. If an inconsistency is encountered flush both `negative`, and `positive`; and go to `backtrack`

Suppose we want to use standard backtracking in combination with `A-3` to solve the n-queens problem. The resulting algorithm is unlikely to be faster than the standard backtracking algorithm, and is probably even slower. This behavior can be explained as follows. The $\neq$ constraint is a negative constraint. There are no positive constraints in the n-queens

problem. Because only positive constraints can compute the values of unknown variables, A-3 will never derive new values, and is thus the same as `partial-test`.

One may wonder why we still want to use `A-3` in case of the n-queens problem, as it is probably slower. `A-3` has two important advantages. Firstly, it allows us to write down the constraints of the problem in a fully declarative way. That is, we only write how the constraints are defined, but we do not specify a specific order of invocation. Secondly, it does not rely on a specific backtrack or choice order. Thus we can use all kinds of backtrack and choice heuristics in combination with data-driven value propagation.

2.5.2 Domain propagation

In order to solve negative constraints, value propagation algorithms, like `A-1`, `A-2`, and `A-3` resort to a generate & test strategy. However, we would like to make active use of both negative and positive constraints. The solution seems to be simple: do not just propagate values but also propagate domains.

In this section we will describe several algorithms that propagate both values and domains. First, however, we would like to say something about the distinction between the value of a variable and its domain. This distinction is not really necessary. The algorithms that will be described in this section use the constraints rules to reduce the domains of the variables. Two special situations can occur. Firstly, a constraint rule removes all the values from the domain of a variable. This means that all the values in this domain are incompatible with the values in domains of the other variables, or the values of the other variables. Secondly, the constraint rule removes all but one value from a domain. The resulting domain is called a *singleton domain*. The value of a variable is known if and only if its domain is a singleton, and the value is unknown if and only if its domain contains more than one value. Furthermore, assigning a value to a variable is the same as reducing its domain to a singleton domain.

`A-3` uses constraint rules to compute the values of unknown variables given the values of other variables. We extend these constraint rules to domain level, in order to be able to propagate domains instead of values.

Example: an extended constraint rule

Consider the following (pseudo) constraint rule.

```
(define rule-65
```

```
(lambda (v1 v2 v3)
  (intersection v3
    (map (lambda (l1 l2)
           (* l1 l2))
         (join v1 v2)))))
```

V1 and v2 are the antecedent variables and v3 is the consequent variable. This rule computes the new domain of v3 given the domains of v1 and v2. The antecedent part of this rule is empty. Suppose that the domain of v1 is (1 2), the domain of v2 is (3 4), and the domain of v3 is (1 2 3 4). The result of the join is

((1 3) (1 4) (2 3) (2 4)),

map returns

(3 4 6 8),

and finally intersection returns,

(3 4).

Notice that because of the intersection operation, the cardinality of the domain of v3 will never increase.

Data-driven domain propagation

B-2 is an extension of A-2, using domain propagation instead of value propagation.

1. shedule. Suppose the domain of variable v_i has been reduced (either because of a choice or an update). Lookup the dependency set D_i of variable v_i and add the constraints rules in D_i to agenda.

2. apply. If the agenda is empty, stop and let choice do its work, else remove a constraint rule, say r_{ij}, from the agenda. Suppose that v_i is the consequent variable of this constraint rule. Lookup the domain d_i, of v_i. Apply constraint rule r_{ij}, and call the result d_i'. If $d_i' \not\subset d_i$ then apply the next constraint rule in agenda. If $d_i' = \emptyset$ then stop, flush the agenda, and backtrack, else update the domain of v_i and go to shedule.

Again we need to change backtrack a little. All the value propagation algorithms, like A-2 or A-3, changed the values of the variables, but the domains always remained the same. Domain propagation algorithms, like B-2, do change the domains of the variables. Backtrack should restore the domains of all the variables that have been updated since the last choice. We propose the following solution.

Whenever choice assign a value to a variable, i.e., all the values but one are removed from the domain of the variable, it will also increase a counter. This counter is denoted by t. Assume all the domains are stored in a list. This list consists of pairs (s_j, d_j) where d_j is the domain of variable v_j, and s_j is a boolean. Whenever a domain d_j needs to be modified, we first check to see if s_j is true. If it is d_j is updated immediately. If, however, s_j is false, then we first cons a pair (t, d_j) onto a reset-list, set s_j to true, and update d_j. If B-2 detects an inconsistency, all the domains on the reset-list that have been added at time t are restored and removed from the reset list, and their s_j are set to false. Now choice can retry the most recent choice point. If choice also fails, i.e, there are no more untried values in the domain of the choice point, then backtrack restores all the domains that have been added at time $t-1$, et cetera.

So far so good. But where do we store the choice points? The answer is simple: on a stack. Let us call this stack the choice point stack. Whenever choice chooses a variable v_i and assigns a value to it, say l_1, three [5] things happen: (1) the counter t is increased, (2) a domain $d_i \backslash \{l_1\}$, is pushed on the choice point stack, and (3) the domain of v_i is reduced to $\{l_1\}$. Whenever choice retries a choice point, it chooses a value from the domain that is on top of the choice point stack. If there are no more values in this domain, then choice stops and fails. Now backtrack will restore the domains and pop the choice point stack. Otherwise, suppose that value l_2 is in the domain. Let v_i be the current variable. Now the following happens, (1) remove value l_2 from the domain that is on top of the choice point stack; (2) increase the counter t; and (3) update the value of v_i, i.e., set its domain to singleton $\{l_2\}$.

Summary

Standard backtracking can be used to solve CSPs. We have described several local propagation algorithms that can be in used combination with (stan-

[5] Actually four things happen because we need to tell the local propagation algorithm which domain has been changed.

dard) backtracking. Local value propagation algorithms do not adequately handle negative constraint. Local domain propagation algorithms can be seen as a generalization of value propagation, and can handle negative constraints. Indeed, propagating values is the same as propagating singleton domains. To be able to implement a domain propagation algorithm several changes had to be made to the backtracking algorithm.

Chapter 3

Modularity, extensibility, and efficiency

One of the main problems of designing finite domain constraint solvers is the tradeoff between generality and efficiency. Most of the systems built so far tend to be application specific and hard to adapt to other, more general tasks [21]. On the other hand the few 'universal' constraint solvers that have been built can be described as "slow". Among others we observed that building fast constraint solvers is not extremely problematic. However, building fast and yet flexible constraint solvers seems to be quite complicated.

3.1 Filters

Recall the assumption that every constraint can be written as a number of constraint rules or extended constraint rules. We have used constraint rules instead of constraints because the distinction between antecedent variables and consequent variables simplified the description of the local propagation algorithms. In this chapter, we focus attention to constraints.

Recall that, a constraint c_i on variables $v_1, \ldots, v_n$ with corresponding domains $d_1, \ldots, d_n$ is a function,

$$c_i : P \rightarrow \{0, 1\},$$

where P denotes the Cartesian product $d_1 \times \cdots \times d_n$. Furthermore we defined a set T,

$$T = \{\boldsymbol{x} \in P | c_i \boldsymbol{x} = 1\}.$$

Let $\prod_i$ denote a projection on variable v_i. We will use $\prod_{1,\ldots,n}$ as a shorthand for $\prod_1, \ldots, \prod_n$. Let F_i be a (mapping) function, that takes as input $d_1, \ldots, d_n$, and returns

$$\prod_{1,\ldots,n} T = d'_1, \ldots, d'_n.$$

We call F_i the *filter* of c_i. Applying a filter to a number of domains is called *filtering*.

Example: a filter

Let `project` be function that takes a set of tuples (generated by `join` and `select`) and projects these tuples on the variables. Example:

```
(project '((1 2 3) (4 5 6) (3 2 1) (6 5 4))),
```

returns

```
((1 4 3 6) (2 5) (3 6 1 4)).
```

The add filter can be defined as,

```
(define add
  (lambda (v1 v2 v3)
    (project
     (select (lambda (val1 val2 val3)
               (= val3 (+ val1 val2)))
             (join v1 v2 v3)))).
```

Describing filters in terms of `project`, `join`, and `select` is not a good idea in practice, as it is rather inefficient. Applying an n-place filter to domains of cardinality c, results in generating and testing c^n tuples, and in the worst case nc^n member operations (in `project`).

By employing the semantics of the constraints, the cost of a filtering can be reduced significantly. Consider the following two implementations of the $\neq$ filter.

```
(define !=
  (lambda (d1 d2)
    (project
     (select (lambda (l1 l2)
               (not (= l1 l2)))
             (join d1 d2)))))
```

```
(define !=
  (lambda (d1 d2)
    (cond ((ground? d1)
           (list d1 (remove (value d1) d2))
          ((ground? d2)
           (list (remove (value d2) d1) d1)
          (else (list d1 d2))))
```

Both filters return a list of, possibly reduced, domains. The first one uses a generate & test approach to reduce the domains. The second implementation actively uses the semantics of the $\neq$ constraint. Notice that the improvement in efficiency can be quite important in case of the $\neq$ filter.

Of course, these examples are trivial. In some cases it is impossible to completely avoid explicit enumeration. In such cases one could resort to a double strategy. First a weak and cheap implicit enumeration method is used to remove some of the incompatible values, and then the expensive enumeration method is used to remove the remaining incompatible values. In case of the add filter, for instance, we can first use interval arithmetic to reduce the boundaries of the domains and the already described generate and test method thereafter.

Notice that the second implementation of the $\neq$ filter does nothing if both domains are non-ground. The first implementation, however, does a lot of needless work.

3.2 Activation and inactivation

Filtering is a powerful, but sometimes costly, domain propagation method. In some cases we know that applying a filter will not lead to a reduction of domains. Also, it may lead to a reduction that is too expensive in terms of computational cost. A sensible approach might be to employ heuristic functions to determine if a filter should be applied. We use two such functions: activation and inactivation conditions. A filter will be applied only if its activation conditions hold, and once its inactivation conditions hold it will never be applied again. Moreover, as in `A-3`, every constraint belongs to a priority class. Let us distinguish two priority classes: `forward` and `lookahead`. Constraint of the `forward` type are given a higher priority. A *primitive constraint* is a structure consisting of:

1. `type`. Every primitive constraint belongs to a priority class.

2. `activation conditions`. If the activation conditions do not hold then application of the filter will not yet lead to a reduction of domains, or possibly a too expensive reduction of domains.

3. `inactivation conditions`. If these hold then further application of the filter will not lead to reduction of domains.

4. `filter`. A mapping function.

Although we don't have to be very strict in this, we say that a primitive constraint is a forward constraint if its activation conditions state that at most one variable is non-ground. The activation conditions of lookahead constraints allow at most two non-ground variables. Without too much effort it can be shown that the inactivation conditions of a forward constraint are the same as the activation conditions. [1] As a consequence, forward constraint 'fire' only once, whereas lookahead constraints may 'fire' several times.

Example: a primitive constraint

```
(define add
  (make-primitive-constraint
    'lookahead                ;;; type
    (lambda (d1 d2 d3)        ;;; activation conditions
      (or  (ground? d1)
           (ground? d2)
           (ground? d3)))
    (lambda (d1 d2 d3)        ;;; inactivation conditions
      (and (ground? d1)
           (ground? d2)
           (ground? d3)))
    (lambda (d1 d2 d3)        ;;; filter
```

[1] This property also causes an extra problem. As the activation and inactivation conditions of forward constraints are the same we have to remember if the forward constraints has already been applied. We use a trace, called activation history, to see which constraint have been applied. A primitive constraint can either be (1) preactive, (2) active or (3) inactive. As soon as the activation conditions of a forward constraint hold, the program first applies the constraint, and then marks the constraint as inactive. On backtracking the program must 'roll-back' the activation history. Although marking the constraints will actually speed up the program, it is not a mathematically elegant solution, as is requires sequencing.

Name	Constraint	Type
Generate & Test	positive & negative	lookback
A-1 & A-2	positive	forward
A-3	positive negative	forward lookback
B-2	positive negative	unconstrained forward
Filtering	positive & negative	unconstrained
Enhanced filtering	negative positive	forward all types
Forward checking	positive & negative	forward
AC-3 Looking ahead	positive & negative	lookahead

Figure 3.1: A characterization of CSP techniques

```
(project
 (select (lambda (l1 l2 l3)
           (= (+ l1 l2) l3))
         (join d1 d2 d3))))))
```

Describing a primitive constraint in terms of activation conditions, inactivation conditions, and filters, is an extremely powerful paradigm as it allows us to classify all the local propagation methods discussed so far. We can even describe generate & test using the primitive constraint paradigm. To do so we distinguish two more priority classes called `lookback` and `unconstrained`. The preconditions of a lookback constraint allow only ground variables. An unconstrained constraint is a primitive constraint without explicit activation conditions.

Figure 3.1, characterizes several CSP techniques. Looking ahead is, more or less, the same as the AC-3 algorithm described in [29]. Forward checking is described in [22]. Enhanced filtering is filtering using primitive constraints.

3.3 Memoization

The third thrashing behavior of standard backtracking was the continual discovery of the same facts. Memoization is a technique that sometimes speeds up computations, or rather prevents computations, by remembering previous results.

3.3.1 Choice memoization

Choice memoization, backmarking [18], or learning [10], has been used in several constraint solvers, e.g., EL and Constraints. It is especially useful in combination with dependency directed backtracking (cf. [12, 15]). In Chapter 2 we described how finite domain constraint solving algorithm can be structured. A backtracking algorithm consists of three elements: `backtrack`, `choice` and a `test`. In the `choice` variables are instantiated and choice points are created. In the `test` part that partial instantiation is tested for local or partial consistency. And `backtrack` tries to restore consistency. If an inconsistency is encountered in the partial instantiation then we can, by analyzing the dependencies between the constraints, try to identify the conflicting instantiations and store these combinations in a set. Let us call this set the nogood set (cf. [15]). A tuple t_1 in the nogood set is called minimal if there exists no other tuple t_2 in the nogood set such that t_1 subsumes t_2. Whenever `choice` instantiates a variable it must check if the resulting partial instantiation does not subsume a minimal nogood [2]. If the (partial) instantiation does subsume a minimal nogood, then the instantiation is inconsistent.

3.3.2 Filter memoization

The cost of finding a solution of a CSP is roughly the cost of the amount of backtracking + the cost of local propagation. The goal of the designer of a constraint solver is of course to minimize the total cost. Determining the right combination of backtracking and local propagation however, can be quite hard, as it is highly problem dependent. Moreover, often it is only possible to increase the execution speed of the constraint solver at the cost of significantly higher memory requirements. Although it pays to use memoization if you want to make the program fast, it will also eat

[2] This can be done by a simple table lookup. Notice that the algorithm can be made incremental fairly easily.

lots of memory. Often it is better to try to improve the efficiency of the program by tuning the activation conditions of the primitive constraints and by reasoning on the semantics of the constraints. Sometimes however there is no alternative and then the following method will certainly work. It must be noted however that it destroys the modularity of the constraint solver. In [33] a similar method is described which is based on ideas in [27].

Let c_i be a constraint on variables $v_1, \ldots, v_n$ with corresponding domains $d_1, \ldots, d_n$, and T,

$$T = \{x \in P | c_i x = 1\}.$$

$M_j(l)$ is the support set of value l of variable v_j and is defined by,

$$M_j(l) = \{(l_1, \ldots, l_{j-1}, l_{j+1}, \ldots, l_n) | (l_1, \ldots, l_{j-1}, l, l_{j+1}, \ldots, l_n) \in T\}.$$

Now the following relation holds

$$F_i(d_1, \ldots, d_n) = (\{l \in d_1 | M_1(l) \neq \emptyset\}, \ldots, \{l \in d_n | M_n(l) \neq \emptyset\}).$$

For every variable in c_i we can build a table of value/support-set pairs and store this table somewhere. If we apply constraint c_i again then we can lookup the support set of every value and variable and check if the supporting values are still in the domains of the support variables. If a support tuple is no longer valid, because at least one support value is no longer in the domain of a support variable, then this tuple is removed from the support set. If the support set becomes empty then the supported value is removed from the domain of the corresponding variable and the table of value/support-set pairs. If an inconsistency is encountered in the local propagation phase then all changes made to the table of value/support-set pairs since the last choice must be made undone [3].

The above described method seems to be very inefficient but can in fact be made very efficient by choosing an appropriate data structure to represent the domains of the variables. A standard method to represent the domains is to use Lisp lists. Unfortunately, this data structure is too inefficient as it can only be accessed sequentially. In [39], a different method is used to represent the domains. Suppose that all the values that were in the initial domain (that is, before filtering) are stored in a vector. Now the domain can be represented as follows. A domain is an object consisting of:

1. Length; An integer indicating the current cardinality of the domain.

[3]This can be done using the reset-list method as described in Chapter 2

2. Boolean vector; The n-th position in this vector is true if the n-th value is in the domain.

3. Mapping functions; Functions that map indices on values and values on indices.

The function to map values on indices can be based on a B-tree or hash table. Now we can define several operations on domains. For example, removing the n-th value from a domain is the same as setting the n-th position in the Boolean vector to false. The intersection operation can be reduced to a bit-wise AND operation.

Using the above described method of storing the domains, filter memoization can be made quite efficient, because instead of storing the values in a support set we can store the indices of the values in the support set. Checking if a support tuple is still valid can now be done in constant time.

Summary

In this chapter we have described several methods to make constraint solvers modular, efficient and extensible. In literature much has been written about choice memoization but little has been said about making active use of the constraint semantics. Although we only gave a short description of this method it should be clear from the example that this method is essential in building fast and yet extensible constraints solvers. The activation/inactivation paradigm has not been described before in literature and is an important contribution as it not only allows us to build efficient but also extensible constraint solvers. Filter memoization has also been proposed by Guesgen [21], in this chapter however we have described a much more efficient method based on a special way to represent the domains.

Chapter 4

Conclusions

Constraints have been studied for three reasons. Firstly, constraints can be used to represent knowledge. Secondly constraints form a computational paradigm. Finally, the constraint approach enables efficient control strategies. The versatility of the constraint approach has been illustrated by describing several application areas.

In the second chapter of this paper we have described, in tutorial fashion, the elements of a finite domain constraint solver based on (standard) backtracking and CSP- or consistency-techniques. [1]

Standard backtracking can be used to solve CSPs. We have described several local propagation algorithms that can be used in combination with (standard) backtracking. Local value propagation does not adequately handle negative constraints. Local domain propagation algorithms can be seen as a generalization of value propagation, and can handle negative constraints.

In the third chapter, which forms the core of this paper, several methods to make constraint solvers modular, efficient and extensible were described. Moreover, we have shown that many of the local propagation methods presented in literature over the last 15 years, have a common basis and can be described elegantly using the activation/inactivation paradigm.

[1] Although not mentioned in this paper other techniques, such as rewriting, breadth-first search, elimination, etc., can be used of course.

Bibliography

[1] R.J. Berndsen and H.A.M. Daniels. Application of constraint propagation in monetary economics. Proceedings of the International conference on expert systems and their applications, Avignon, 1988.

[2] W. Bibel. Constraint Satisfaction from a Deductive Viewpoint. Artificial Intelligence, Vol. 35, 1988.

[3] D.G. Bobrow (ed.). Qualitative reasoning about physical systems: An introduction. Artificial Intelligence, Vol. 24, 1984.

[4] A. Borning. The Programming Language Aspects of THINGLAB, a Constraint-Oriented Simulation Laboratory. ACM Transactions on Programming Languages and Systems, Vol.3, No. 4, 1981.

[5] J.M. Broek and H.A.M. Daniels. Application of Constraint Logic Programming to ALM modeling. Proceedings of Cecoia 2, Conference on Economics and Artificial Intelligence, Paris, 1990.

[6] M. Bruynooghe. Solving Combinatorial Search Problems by Intelligent Backtracking. Information Processing Letters, 12(1), 1981.

[7] A. Colmerauer. Note sur Prolog III. in Actes du Seminaire 1986 - Programmation en Logique, Tregastel, 1986. (in French)

[8] E. Davis. Constraint Propagation with Interval Labels. Artificial Intelligence, Vol. 32, 1987.

[9] R. Davis. Diagnostic Reasoning based on Structure and Behavior. Artificial Intelligence, Vol. 24, 1984.

[10] R. Dechter. Enhancement Schemes for Constraint Processing: Backjumping, Learning and Cutset Decomposition. Artificial Intelligence, Vol. 41, 1990.

[11] R. Dechter and J. Pearl. Network-Based Heuristics for Constraint-Satisfaction Problems. Artificial Intelligence, Vol. 34, 1988.

[12] J. de Kleer. A Comparison of ATMS and CSP Techniques. Proceedings IJCAI, Detroit, 1989.

[13] J. de Kleer and B.C. Williams. Diagnosing Multiple Faults. Artificial Intelligence, Vol 32, 1987.

[14] J. de Kleer and J.S. Brown. A Qualitative Physics based on Confluences. Artificial Intelligence, Vol. 24, 1984.

[15] J. Doyle. A Truth Maintenance System. Artificial Intelligence, Vol 12, 1979.

[16] A. Feelders. Explanation and Diagnosis in financial models of the firm: An implementation in CHIP. Technical Report, ECRC, 1990.

[17] B.J. Freeman-Benson, J. Malony, and A. Borning. An Incremental Constraint Solver. Communications of the ACM, Vol. 33, No 1, 1990.

[18] J. Gaschnig. Performance measurement and analysis of certain search algorithms. Technical Report, CMU-CS-79-124, Carnegie Mellon University, 1979.

[19] J. Gosling. Algebraic Constraints. Ph.D. dissertation, Carnegie-Mellon University, 1983.

[20] T. Graf, P. Van Hentenryck, C. Pradelles, and L. Zimmer. Simulation of Hybrid Circuits in Constraint Logic Programming. Proceedings IJCAI, Detroit, 1989.

[21] H.W. Guesgen. A Universal Constraint Programming Language. Proceedings IJCAI, Detroit, 1989.

[22] R.M. Haralick and G.L. Elliot. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. Artificial Intelligence, Vol. 14, 1980.

[23] J. Jaffar and J-L. Lassez. Constraint Logic Programming. Proceedings of the 14th ACM POPL Symposium, Munich, 1987.

[24] J. Jaffar and S. Michaylov. Methodology and Implementation of a CLP system. Proceedings of the Fourth International Conference on Logic Programming, Melbourne, 1987. (MIT Press)

[25] D.E. Knuth. The METAFONT book, volume C of Computers and Typesetting. Addison-Wesley, Reading, MA, 1986.

[26] B. Kuipers. Qualitative Simulation. Artificial Intelligence, Vol. 29, 1986.

[27] J-L. Lauriere. A Language and a Program for Stating and Solving Combinatorial Problems. Artificial Intelligence, Vol. 10, 1978.

[28] Wm. Leler. Constraint Programming Languages: Their Specification and Generation. Addison Wesley, Reading, MA, 1988.

[29] A.K. Mackworth. Consistency in Networks of Relations. Artificial Intelligence, Vol. 8, 1977.

[30] A.K. Mackworth and E.C. Freuder. The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems. Artificial Intelligence, Vol. 25, 1985.

[31] A.K. Mackworth. Constraint Satisfaction. in S. Shapiro (ed.), The Encyclopedia of Artificial Intelligence, Wiley, New York, 1987.

[32] P. Meseguer. Constraint Satisfaction Problems: An Overview. AICOM, Vol. 2, No. 1, 1989.

[33] R. Mohr and T.C. Henderson. Arc and Path Consistency Revisited. Artificial Intelligence, Vol. 28, 1986.

[34] U. Montanari. Networks of Constraints: Fundamental Properties and Applications to Picture Processing. Information Sciences, Vol. 7, 1974.

[35] R.M. Stallman and G.J. Sussman. Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis. Artificial Intelligence, Vol. 9, 1977.

[36] G.L. Steele. The Definition and Implementation of a Computer Programming Language Based on Constraints. Ph.D. Thesis, Mass. Institute of Technology, 1980.

[37] G.J. Sussman and G.L. Steele. CONSTRAINTS- A language for Expressing Almost-Hierarchial Descriptions. Artificial Intelligence, Vol. 14, 1980.

[38] J.E. Sutherland. SKETCHPAD: A man-machine graphical communication system. Technical Report 296, MIT Lincoln Labs, Cambridge, MA, 1963.

[39] P. Van Hentenryck. Constraint Satisfaction in Logic Programming. MIT Press, Cambridge, MA, 1990.

[40] D. Waltz. Understanding line drawings of scenes with shadows. in The Psychology of Computer Vision. P. Winston (ed.). McGraw-Hill, New York, 1975.

[41] C.J. van Wyk. A Language for Typesetting Graphics. Ph.D. dissertation, Stanford University, 1980.

[42] M. Zweben and M. Eskey. Constraint Satisfaction with Delayed Evaluation. Proceedings IJCAI, Detroit, 1989.

ITK: P.O. BOX 90153 5000 LE TILBURG THE NETHERLANDS